unir LA UNIVERSIDAD EN INTERNET

Técnicas de Inteligencia Artificial

Actividad 2. Trabajando con redes neuronales y Deep Learning

Presentado por: Hugo Rodríguez Donato

Fecha: 18/01/2025

Importación de librerias necesarias

In [1]:
```
# Para esta actividad se importarán las siguientes librerías:
# Pandas, Numpy, Matplotlib, Seaborn, Scikit-learn tensorflow
import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
import seaborn as sns

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
```

Parte II. Clasificación

Cargar el Dataset

Con al menos 1000 instancias, una variable/atributo de la salida, y que dependa de, al menos, 6 variables/atributos de entrada.

In [ ]:
```
#Código para cargar el Dataset

url = 'https://raw.githubusercontent.com/Hugord7/Tecnicas-Inteligencia-Artificial/refs/heads/main/updated_pollution_dataset.cs
```

Descripción de la fuente del Dataset

Haga una descripción de la fuente de dayos utilizada (Incluya los enlaces necesarios).

El dataset escogido se centra en la evaluación de la calidad del aire en varias regiones. El dataset contiene 5000 muestras y captura factores ambientales y demográficos críticos que influyen en los niveles de contaminación.

Enlace a Kaggle del dataset: https://www.kaggle.com/datasets/mujtabamatin/air-quality-and-pollution-assessment

Explique el problema a resolver.

Descripción del problema. Tipo de problema (justifique). Variable objetivo, variables de entrada. Utilidad de su posible solución. Elementos adicionales que considere relevantes (no son necesarios contenidos teóricos, sino explicar qué relaciones tratas de comprobar y con qué métodos).

El problema a resolver es saber predecir la calidad del aire según los datos del entorno. Es decir, se usará un algoritmo de clasificación, el cual con los atributos de entrada (humedad, temperatura...) podrá predecir si la calidad del aire es buena o mala (variable objetivo). Esta solución se podría implementar en las ciudades y pueblos, además de en edificios particulares, o incluso casas, ya que gracias a este sistema se podría saber a tiempo real si el aire que estamos respirando es bueno o malo para nosotros, y poder aplicar soluciones instantáneas como ventilar la habitación, salir del edificio, o como se hace en las grandes ciudades, restringir el uso de los coches más contaminantes.

Caracterización del Dataset

Realice una descripción de los datos con:

> - Número de instancias en total.
> - Número de atributos de entrada, su significado y tipo.

> - Número de clases de la variable objetivo, indicando que representan dichas clases y el tipo de valor que toman.
> - Número de instancias pertenecientes a cada clase en la variable objetivo.
> - Estadísticas de la variable objetivo.
> - Estadísticas los atributos en relación con la variable objetivo.

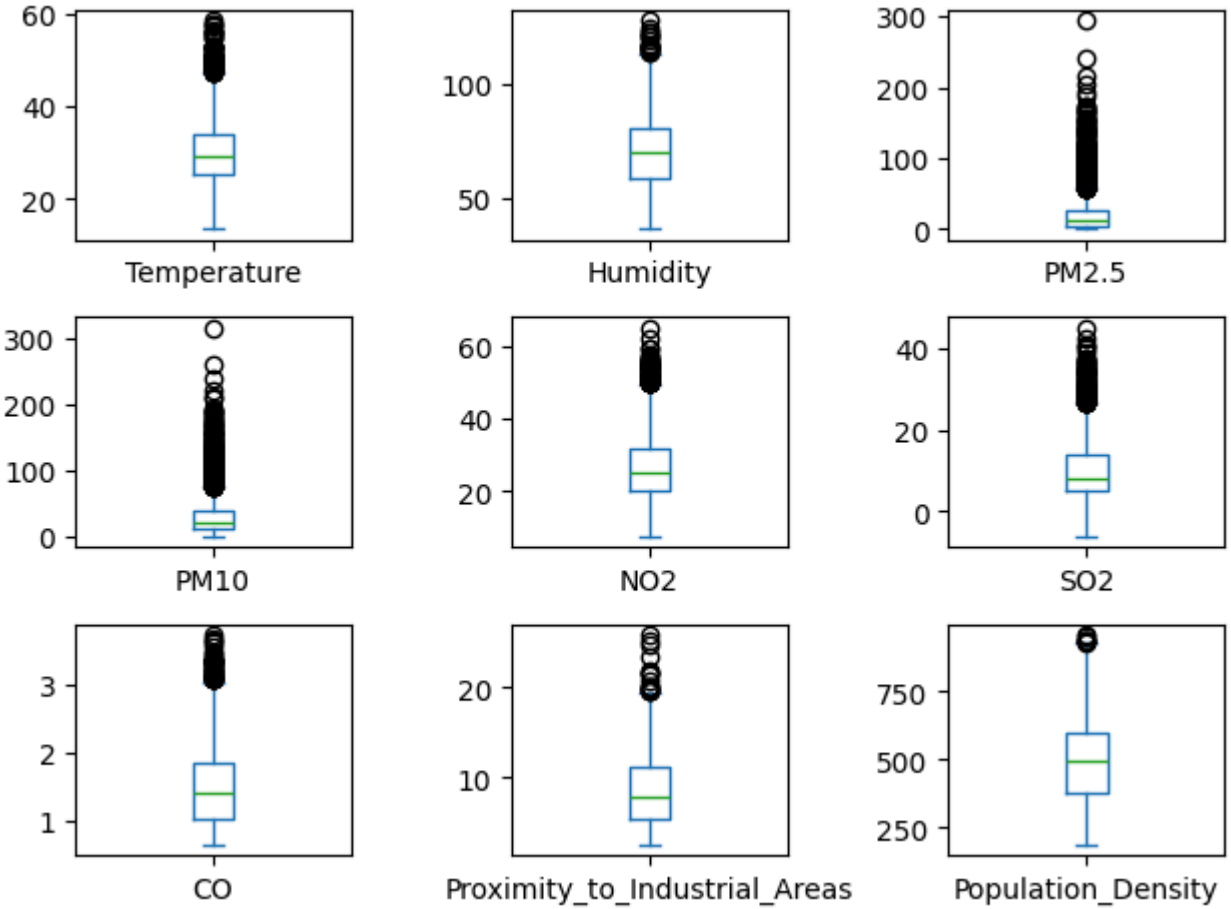
Se incorporá una pequeña descripción (EDA) del conjunto de datos utilizado. Se analiza el dataset proporcionando, se muestra al menos algunas de sus características mediante tablas y al menos algunas de ellas en modo gráfico (p.ej., histogramas, diagramas de dispersión, diagramas de cajas y bigotes, etc.)

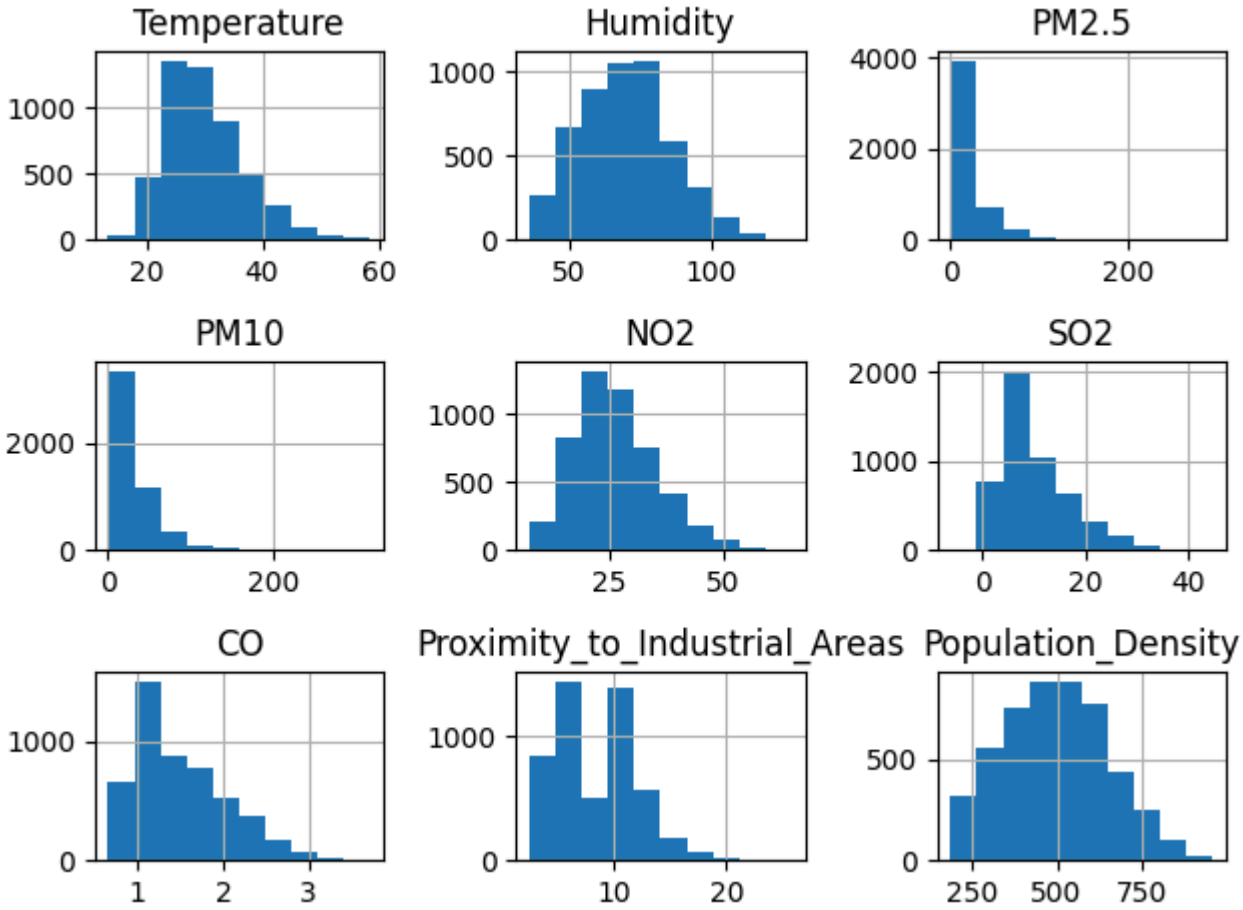
In [3]:
```
#Código que responde a la descripción anterior
# Código que responde a la descripción anterior
from pandas import read_csv

# Guardamos el dataset en df leyendo el csv sabiendo que esta separado por ','
df = pd.read_csv( url, sep=',', on_bad_lines='skip')
#df=pd.read_csv(excel_path_file, sep=',')

# Imprimimos la cabezera del dataset para hacernos una idea del contenido de este
print('Cabecera Dataset')
print(df.head(5))
print('----------------')

# Con el comando 'shape' podemos ver cuantas instancias y atributos hay en total
print('Instancias y atributos Dataset')
print(df.shape)
print('----------------')

# Con el comando 'describe' analizamos el dataset de una forma general, observando los valores de cada columna
print('Descripción Dataset')
print(df.describe())
print('----------------')

# Con el comando 'groupby' analizamos los valores de la clase 'class'
print('Clasificación tipo clases Dataset')
print(df.groupby('Air Quality').size())
```

```
Cabecera Dataset
   Temperature  Humidity  PM2.5  PM10   NO2   SO2    CO  \
0         29.8      59.1    5.2  17.9  18.9   9.2  1.72
1         28.3      75.6    2.3  12.2  30.8   9.7  1.64
2         23.1      74.7   26.7  33.8  24.4  12.6  1.63
3         27.1      39.1    6.1   6.3  13.5   5.3  1.15
4         26.5      70.7    6.9  16.0  21.9   5.6  1.01

   Proximity_to_Industrial_Areas  Population_Density Air Quality
0                            6.3                 319    Moderate
1                            6.0                 611    Moderate
2                            5.2                 619    Moderate
3                           11.1                 551        Good
4                           12.7                 303        Good
----------------
Instancias y atributos Dataset
(5000, 10)
----------------
Descripción Dataset
       Temperature     Humidity        PM2.5         PM10          NO2  \
count  5000.000000  5000.000000  5000.000000  5000.000000  5000.000000
mean     30.029020    70.056120    20.142140    30.218360    26.412100
std       6.720661    15.863577    24.554546    27.349199     8.895356
min      13.400000    36.000000     0.000000    -0.200000     7.400000
25%      25.100000    58.300000     4.600000    12.300000    20.100000
50%      29.000000    69.800000    12.000000    21.700000    25.300000
75%      34.000000    80.300000    26.100000    38.100000    31.900000
max      58.600000   128.100000   295.000000   315.800000    64.900000

              SO2           CO  Proximity_to_Industrial_Areas  \
count  5000.000000  5000.000000                    5000.000000
mean     10.014820     1.500354                       8.425400
std       6.750303     0.546027                       3.610944
min      -6.200000     0.650000                       2.500000
25%       5.100000     1.030000                       5.400000
50%       8.000000     1.410000                       7.900000
75%      13.725000     1.840000                      11.100000
max      44.900000     3.720000                      25.800000

       Population_Density
count         5000.000000
```

```
mean           497.423800
std            152.754084
min            188.000000
25%            381.000000
50%            494.000000
75%            600.000000
max            957.000000
----------------
Clasificación tipo clases Dataset
Air Quality
Good         2000
Hazardous     500
Moderate     1500
Poor         1000
dtype: int64
```

In [6]:
```python
# Código que responde a la descripción anterior (incorpore las lineas de code necesarias. Describa cadas sentencia de código)
from pandas.plotting import scatter_matrix

# Mostramos el diagrama de cajas y bigotes
df.plot(kind='box', subplots=True, layout=(3, 3), sharex=False, sharey=False)
plt.tight_layout()
plt.show()

# # Mostramos el historigrama
df.hist()
plt.tight_layout()
plt.show()
```


Temperature
Humidity
PM2.5
PM10
NO2
SO2
CO
Proximity_to_Industrial_Areas
Population_Density



En este apartado vemos que nuestro dataset se compone de 10 atributos, el último de ellos considerado atributo de salida (clase), el cual tiene cuatro estados ('Good', 'Hazardous', 'Moderate', 'Poor') que indican la calidad del aire.

Good: Bueno. Hazardous: Peligroso. Moderate: Moderado. Poor: Pobre.

Observamos 5000 instancias, de las cuales 2000 pertenecen a 'Good', 1500 a 'Moderate', 1000 a 'Poor' y 500 a 'Hazardous'.

Los 9 atributos de entrada son los siguientes Temperature (temperatura), Humidity (humedad), PM2.5 (concentración ($\mu g/m^3$): niveles de partículas finas), NO2 (concentración de NO2 (ppb): niveles de dióxido de nitrógeno), PM10 (concentración de PM10 ($\mu g/m^3$): niveles de partículas gruesas), SO2 (concentración de SO2 (ppb): niveles de dióxido de azufre), CO (concentración de CO (ppm): niveles de monóxido de

carbono), Proximity_to_Industrial_Areas (distancia a la zona industrial más cercana) y Population_Density (número de habitantes por kilómetro cuadrado en la región). Todos ellos son atributos numéricos.

Como se comprueba en el siguiente apartado, no hay valores nulos en el dataset.

En un par de párrafos haga un resumen de los principales hallazagos encontrados:

Por lo que se puede observar, intuimos que el algoritmo podría ser mejor a la hora de evaluar aire de buena calidad, ya que es donde más datos se pueden entrenar. Se evalúan sitios no muy cercanos a áreas industriales, con una media de población de 497 personas por kilometro cuadrado. También vemos como la temperatura media es de 30 grados, siendo el rango en el que se encuentran más datos entre 20 y 40 grados.

Preprocesamiento del dataset. Transformaciones previas necesarias para la modelación

In [8]:
```
# Código que realice las transformaciones necesarias para poder realizar los procesos de modelación. Ej.One hot enconding
df.isna().sum()
```

Out[8]:
```
Temperature                    0
Humidity                       0
PM2.5                          0
PM10                           0
NO2                            0
SO2                            0
CO                             0
Proximity_to_Industrial_Areas  0
Population_Density             0
Air Quality                    0
dtype: int64
```

División del dataset en datos de entrenamiento y datos de test

In [9]:
```
# Código que realice la división en entrenamiento y test, de acuerdo con la estrategia de evaluación planeada. Describa cuál e
# En este apartado se divide el dataset en 80% datos entrenamiento y 20% datos de test. Hay 9 columnas, de las cuales 8 de ell
# y una (salary_in_usd) es atributo de salida (clase)
```

```
# Importamos las funciones y metodos a evaluar desde Scikit-learn
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.metrics import mean_squared_error


from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier

# (División de datos para modelos sin red neuronal)Se divide el dataset en 80% datos entrenamiento y 20% datos test ya que
# disponemos de pocos datos para el entrenamiento y evaluación, por esta razón se escoge un número alto de datos de entrenamie

array = df.values
X = array[:,0:9]
y = array[:,9]
X_train, X_validation, Y_train, Y_validation = train_test_split(X, y, test_size=0.20, random_state=1, shuffle=True)

# (División de datos para modelos con red neuronal) Se divide el dataset en 80% datos entrenamiento y 20% datos test ya que
# disponemos de pocos datos para el entrenamiento y evaluación, por esta razón se escoge un número alto de datos de entrenamie

train_dataset = df.sample(frac=0.8,random_state=0)
test_dataset = df.drop(train_dataset.index)

# Además en el algoritmo de red neuronal se aplica el cambio de atributo categórico a numérico para su posterior entrenamiento
train_dataset["Air Quality"].replace({'Hazardous': 0, 'Poor': 1, 'Moderate': 2, 'Good': 3}, inplace=True)
test_dataset["Air Quality"].replace({'Hazardous': 0, 'Poor': 1, 'Moderate': 2, 'Good': 3}, inplace=True)
```

```
C:\Users\hugor\AppData\Local\Temp\ipykernel_23908\3241634455.py:32: FutureWarning: A value is trying to be set on a copy of a D
ataFrame or Series through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are set
ting values always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] = d
f[col].method(value) instead, to perform the operation inplace on the original object.


  train_dataset["Air Quality"].replace({'Hazardous': 0, 'Poor': 1, 'Moderate': 2, 'Good': 3}, inplace=True)
C:\Users\hugor\AppData\Local\Temp\ipykernel_23908\3241634455.py:32: FutureWarning: Downcasting behavior in `replace` is depreca
ted and will be removed in a future version. To retain the old behavior, explicitly call `result.infer_objects(copy=False)`. To
opt-in to the future behavior, set `pd.set_option('future.no_silent_downcasting', True)`
  train_dataset["Air Quality"].replace({'Hazardous': 0, 'Poor': 1, 'Moderate': 2, 'Good': 3}, inplace=True)
C:\Users\hugor\AppData\Local\Temp\ipykernel_23908\3241634455.py:33: FutureWarning: A value is trying to be set on a copy of a D
ataFrame or Series through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work because the intermediate object on which we are set
ting values always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using 'df.method({col: value}, inplace=True)' or df[col] = d
f[col].method(value) instead, to perform the operation inplace on the original object.


  test_dataset["Air Quality"].replace({'Hazardous': 0, 'Poor': 1, 'Moderate': 2, 'Good': 3}, inplace=True)
C:\Users\hugor\AppData\Local\Temp\ipykernel_23908\3241634455.py:33: FutureWarning: Downcasting behavior in `replace` is depreca
ted and will be removed in a future version. To retain the old behavior, explicitly call `result.infer_objects(copy=False)`. To
opt-in to the future behavior, set `pd.set_option('future.no_silent_downcasting', True)`
  test_dataset["Air Quality"].replace({'Hazardous': 0, 'Poor': 1, 'Moderate': 2, 'Good': 3}, inplace=True)
```

Propuesta de arquitectura de red neuronal

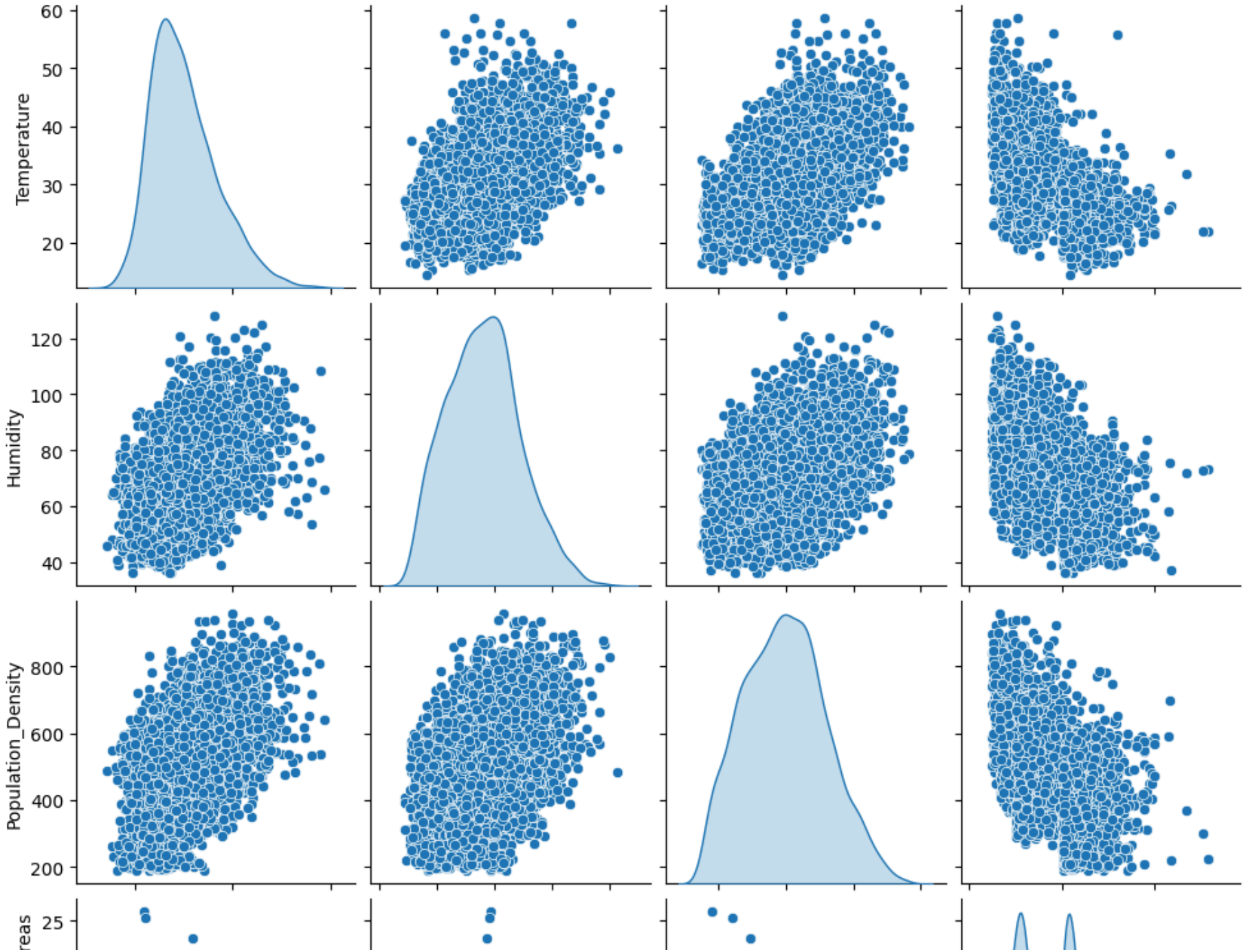
Describe:

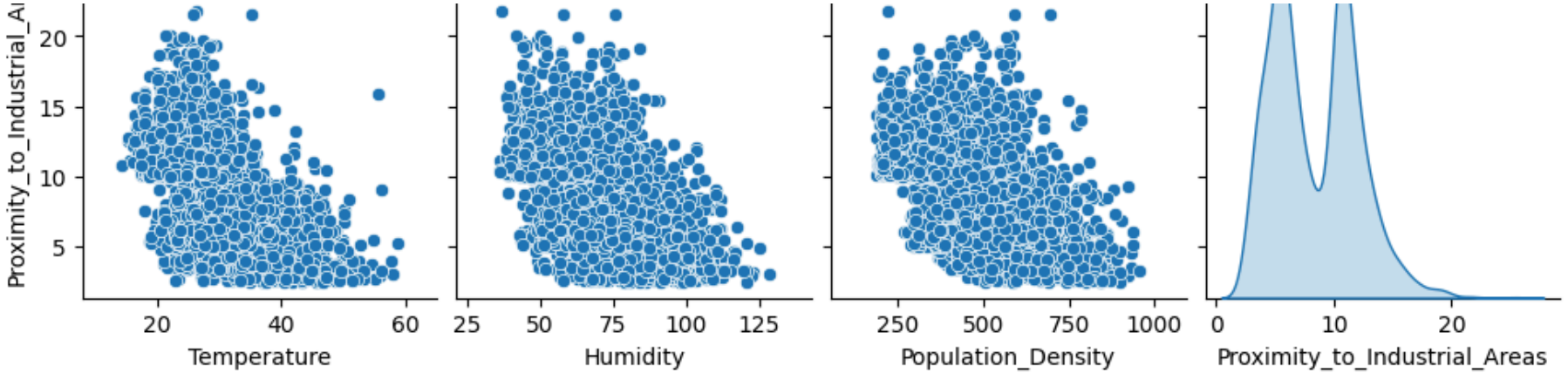
- las neuronas en la capa de entrada
- las capas intermedias – al menos dos –
- capa de salida
- funciones de activación

Al menos utiliza relu en algunas de las capas intermedias y utiliza softmax en la capa de salida.

In [10]:
```
sns.pairplot(train_dataset[["Temperature", "Humidity", "Population_Density", "Proximity_to_Industrial_Areas"]], diag_kind="kde
```

Out[10]:
```
<seaborn.axisgrid.PairGrid at 0x22a3eba3170>
```


Temperature
Humidity
Population_Density
60
50
40
30
20
120
100
80
60
40
800
600
400
200
25



In [11]:
```
# Quitamos la columna 'Air Quality' de los datos 'train_stats'.

train_stats = train_dataset.describe()
train_stats.pop("Air Quality")
train_stats = train_stats.transpose()
```

In [12]:
```
# Se extrae la columna que queremos predecir (Air Quality) de los datos de entrenamiento y se asigna a las variables 'train_la

train_labels = train_dataset.pop('Air Quality')
test_labels = test_dataset.pop('Air Quality')
```

In [13]:
```
# Código de la estructuración de la red

def build_model():
   model = keras.Sequential([
     layers.Dense(64,input_shape=(9,)),
     layers.Dense(128, activation="relu"),
     layers.Dense(128, activation="relu"),
     layers.Dense(4, activation="softmax")
   ])
   model.compile(optimizer="adam",
                 loss="sparse_categorical_crossentropy",
                 metrics=["accuracy"])
   return model
```

In [14]:
```
# Código de la inspección del modelo de red
model = build_model()
model.summary()
```

```
c:\Users\hugor\AppData\Local\Programs\Python\Python312\Lib\site-packages\keras\src\layers\core\dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 64)	640
dense_1 (Dense)	(None, 128)	8,320
dense_2 (Dense)	(None, 128)	16,512
dense_3 (Dense)	(None, 4)	516

Total params: 25,988 (101.52 KB)

Trainable params: 25,988 (101.52 KB)

Non-trainable params: 0 (0.00 B)

Como se puede observar el modelo consta de 4 capas. Una capa de entrada, dos capas intermedias y una última capa final.

La capa de entrada la forman 64 neuronas. Las dos capas intermedias tienen 128 neuronas cada una. La última capa cuenta con 4 neuronas (las 4 clases de salida)

La función de activación usada es 'relu' para las capas intermedias, su propósito principal es no introducir linealidad en el modelo, lo cual es esencial para aprender relaciones complejas en los datos. Para la capa de salida la función de activación es 'softmax', la cual se usa en problemas de clasificación multiclase, la cual nos da la probabilidad de las clases posibles (la suma de las probabilidades de todas las clases suma 1).

Ajuste de modelo de Clasificación RNA

Mediante Python y utilizando al menos Keras sobre TensorFlow 2.0 (tensorflow.keras), entrena el modelo o modelos de red neuronal escogidos.

In [15]:
```
# Código de ajuste y entrenamiento
# Entrenamos el modelo:
cv_results_3 = model.fit(train_dataset, train_labels, epochs=200)

# Guardamos el resultado 'Accuracy' en la variable 'cv_results_3_graph', para después poder graficarla y compararla con el res
cv_results_3 = cv_results_3.history['accuracy']
```

```
Epoch 1/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.3575 - loss: 10.4629
Epoch 2/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5811 - loss: 1.6218
Epoch 3/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5722 - loss: 1.3331
Epoch 4/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.6698 - loss: 0.9039
Epoch 5/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7254 - loss: 0.7218
Epoch 6/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.6680 - loss: 0.9900
Epoch 7/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7160 - loss: 0.8084
Epoch 8/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7064 - loss: 0.8996
Epoch 9/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7500 - loss: 0.6360
Epoch 10/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7377 - loss: 0.7101
Epoch 11/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7541 - loss: 0.6088
Epoch 12/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7863 - loss: 0.4955
Epoch 13/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.7503 - loss: 0.6453
Epoch 14/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7327 - loss: 0.7578
Epoch 15/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.7819 - loss: 0.5476
Epoch 16/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7591 - loss: 0.5592
Epoch 17/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7816 - loss: 0.5461
Epoch 18/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7759 - loss: 0.5280
Epoch 19/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7935 - loss: 0.5127
Epoch 20/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7852 - loss: 0.5201
Epoch 21/200
```

```
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.7901 - loss: 0.4685
Epoch 22/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8008 - loss: 0.4888
Epoch 23/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8112 - loss: 0.4634
Epoch 24/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7907 - loss: 0.4933
Epoch 25/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8282 - loss: 0.4357
Epoch 26/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8174 - loss: 0.4591
Epoch 27/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7921 - loss: 0.4836
Epoch 28/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8227 - loss: 0.4356
Epoch 29/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7923 - loss: 0.4856
Epoch 30/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8166 - loss: 0.4645
Epoch 31/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8054 - loss: 0.4560
Epoch 32/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8148 - loss: 0.4367
Epoch 33/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8335 - loss: 0.4255
Epoch 34/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8250 - loss: 0.4276
Epoch 35/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8170 - loss: 0.4339
Epoch 36/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8387 - loss: 0.4134
Epoch 37/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8343 - loss: 0.4129
Epoch 38/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8419 - loss: 0.4008
Epoch 39/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8368 - loss: 0.4017
Epoch 40/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8321 - loss: 0.4070
Epoch 41/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8246 - loss: 0.4019
```

```
Epoch 42/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8303 - loss: 0.4010
Epoch 43/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8492 - loss: 0.3858
Epoch 44/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8364 - loss: 0.4036
Epoch 45/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8419 - loss: 0.3822
Epoch 46/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8353 - loss: 0.3961
Epoch 47/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8313 - loss: 0.3908
Epoch 48/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8455 - loss: 0.3761
Epoch 49/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8507 - loss: 0.3860
Epoch 50/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8445 - loss: 0.3888
Epoch 51/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8547 - loss: 0.3613
Epoch 52/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8320 - loss: 0.4067
Epoch 53/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8505 - loss: 0.3692
Epoch 54/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8486 - loss: 0.3890
Epoch 55/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8429 - loss: 0.3889
Epoch 56/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8563 - loss: 0.3547
Epoch 57/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8522 - loss: 0.3587
Epoch 58/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8340 - loss: 0.3894
Epoch 59/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8635 - loss: 0.3560
Epoch 60/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8691 - loss: 0.3239
Epoch 61/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8660 - loss: 0.3369
Epoch 62/200
```

```
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8589 - loss: 0.3583
Epoch 63/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8524 - loss: 0.3429
Epoch 64/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8754 - loss: 0.3360
Epoch 65/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8643 - loss: 0.3351
Epoch 66/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8591 - loss: 0.3554
Epoch 67/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8661 - loss: 0.3346
Epoch 68/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.8737 - loss: 0.3131
Epoch 69/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8776 - loss: 0.3178
Epoch 70/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8711 - loss: 0.3215
Epoch 71/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8736 - loss: 0.3174
Epoch 72/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8689 - loss: 0.3203
Epoch 73/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8709 - loss: 0.3110
Epoch 74/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8622 - loss: 0.3473
Epoch 75/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8661 - loss: 0.3408
Epoch 76/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8827 - loss: 0.3115
Epoch 77/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8726 - loss: 0.3254
Epoch 78/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8839 - loss: 0.2957
Epoch 79/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8848 - loss: 0.3057
Epoch 80/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.8989 - loss: 0.2719
Epoch 81/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8840 - loss: 0.2882
Epoch 82/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8807 - loss: 0.2962
```

```
Epoch 83/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8852 - loss: 0.2918
Epoch 84/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8923 - loss: 0.2687
Epoch 85/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8772 - loss: 0.2979
Epoch 86/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8784 - loss: 0.3168
Epoch 87/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8815 - loss: 0.2764
Epoch 88/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8912 - loss: 0.2751
Epoch 89/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9004 - loss: 0.2503
Epoch 90/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8956 - loss: 0.2623
Epoch 91/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9035 - loss: 0.2461
Epoch 92/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8971 - loss: 0.2518
Epoch 93/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9071 - loss: 0.2559
Epoch 94/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8953 - loss: 0.2542
Epoch 95/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9007 - loss: 0.2447
Epoch 96/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8849 - loss: 0.2867
Epoch 97/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8908 - loss: 0.2572
Epoch 98/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.9037 - loss: 0.2626
Epoch 99/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9044 - loss: 0.2446
Epoch 100/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8880 - loss: 0.2553
Epoch 101/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9115 - loss: 0.2382
Epoch 102/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9040 - loss: 0.2382
Epoch 103/200
```

```
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8939 - loss: 0.2460
Epoch 104/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9052 - loss: 0.2415
Epoch 105/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9245 - loss: 0.1984
Epoch 106/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9191 - loss: 0.2186
Epoch 107/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9275 - loss: 0.2019
Epoch 108/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9056 - loss: 0.2298
Epoch 109/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9142 - loss: 0.2228
Epoch 110/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9079 - loss: 0.2279
Epoch 111/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9268 - loss: 0.2097
Epoch 112/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9200 - loss: 0.2087
Epoch 113/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9256 - loss: 0.1968
Epoch 114/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9222 - loss: 0.2003
Epoch 115/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9087 - loss: 0.2427
Epoch 116/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9090 - loss: 0.2224
Epoch 117/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9138 - loss: 0.2247
Epoch 118/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9077 - loss: 0.2353
Epoch 119/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9219 - loss: 0.1969
Epoch 120/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9302 - loss: 0.1823
Epoch 121/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9158 - loss: 0.2098
Epoch 122/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9044 - loss: 0.2481
Epoch 123/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9027 - loss: 0.2423
```

```
Epoch 124/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9236 - loss: 0.1928
Epoch 125/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9229 - loss: 0.1864
Epoch 126/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9108 - loss: 0.2146
Epoch 127/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9178 - loss: 0.2049
Epoch 128/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9225 - loss: 0.1960
Epoch 129/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9241 - loss: 0.1980
Epoch 130/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9329 - loss: 0.1781
Epoch 131/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9284 - loss: 0.1896
Epoch 132/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9172 - loss: 0.1960
Epoch 133/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9279 - loss: 0.1885
Epoch 134/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9351 - loss: 0.1658
Epoch 135/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9289 - loss: 0.1746
Epoch 136/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9317 - loss: 0.1756
Epoch 137/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9279 - loss: 0.1763
Epoch 138/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9115 - loss: 0.2414
Epoch 139/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9121 - loss: 0.2212
Epoch 140/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9163 - loss: 0.2084
Epoch 141/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9163 - loss: 0.1997
Epoch 142/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9216 - loss: 0.1933
Epoch 143/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9268 - loss: 0.1904
Epoch 144/200
```

```
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9271 - loss: 0.1799
Epoch 145/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9233 - loss: 0.1934
Epoch 146/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9392 - loss: 0.1722
Epoch 147/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9373 - loss: 0.1739
Epoch 148/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9198 - loss: 0.2000
Epoch 149/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9296 - loss: 0.1842
Epoch 150/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9321 - loss: 0.1822
Epoch 151/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9040 - loss: 0.2432
Epoch 152/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9374 - loss: 0.2178
Epoch 153/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9110 - loss: 0.2233
Epoch 154/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9344 - loss: 0.1733
Epoch 155/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9143 - loss: 0.1969
Epoch 156/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9362 - loss: 0.1793
Epoch 157/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9267 - loss: 0.1824
Epoch 158/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9353 - loss: 0.1617
Epoch 159/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9331 - loss: 0.1751
Epoch 160/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9229 - loss: 0.1890
Epoch 161/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9296 - loss: 0.1749
Epoch 162/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9114 - loss: 0.2108
Epoch 163/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9306 - loss: 0.1792
Epoch 164/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9334 - loss: 0.1719
```

```
Epoch 165/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9222 - loss: 0.1921
Epoch 166/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9319 - loss: 0.1694
Epoch 167/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9181 - loss: 0.1870
Epoch 168/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9357 - loss: 0.1510
Epoch 169/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9420 - loss: 0.1565
Epoch 170/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9264 - loss: 0.1753
Epoch 171/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9285 - loss: 0.1779
Epoch 172/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9305 - loss: 0.1719
Epoch 173/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9067 - loss: 0.2239
Epoch 174/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9171 - loss: 0.2014
Epoch 175/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9339 - loss: 0.1679
Epoch 176/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9287 - loss: 0.1780
Epoch 177/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9250 - loss: 0.1902
Epoch 178/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9271 - loss: 0.1753
Epoch 179/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9354 - loss: 0.1634
Epoch 180/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9448 - loss: 0.1556
Epoch 181/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9231 - loss: 0.1846
Epoch 182/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9346 - loss: 0.1700
Epoch 183/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9447 - loss: 0.1457
Epoch 184/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9310 - loss: 0.1651
Epoch 185/200
```

```
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9210 - loss: 0.2002
Epoch 186/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9291 - loss: 0.1773
Epoch 187/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9232 - loss: 0.1868
Epoch 188/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9257 - loss: 0.1875
Epoch 189/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9476 - loss: 0.1460
Epoch 190/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9219 - loss: 0.1899
Epoch 191/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9412 - loss: 0.1598
Epoch 192/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9419 - loss: 0.1567
Epoch 193/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9344 - loss: 0.1715
Epoch 194/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9267 - loss: 0.1767
Epoch 195/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9276 - loss: 0.1752
Epoch 196/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9278 - loss: 0.1715
Epoch 197/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9336 - loss: 0.1662
Epoch 198/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9394 - loss: 0.1623
Epoch 199/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9298 - loss: 0.1826
Epoch 200/200
125/125 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9346 - loss: 0.1660
```

Evaluación de modelo RNA

Defina las estadísticas (métricas) de evaluación, y dividiendo el dataset en datos de entrenamiento, validación y datos de test prueba tu propuesta.

Visualice el progreso de entrenamiento del modelo y muestre las estadísticas de evaluación para los conjuntos de entrenamiento y validación.

In [16]:
```
# Código de evaluación de la red propuesta (entrenamiento y validación)
media = np.mean(cv_results_3)
print("Media:", media)

max_val = np.max(cv_results_3)
print("Valor más alto:", max_val)

train_loss, train_acc = model.evaluate(train_dataset, train_labels,
verbose=2)
print("\nTrain accuracy:", train_acc,train_loss)
```

```
Media: 0.8765150007605552
Valor más alto: 0.940500020980835
125/125 - 0s - 2ms/step - accuracy: 0.9380 - loss: 0.1622

Train accuracy: 0.9380000233650208 0.1621793657541275
```

Evalúe los resultados para el conjunto de test.

In [18]:
```
# Código de evaluación de la red propuesta (evaluación conjunto de test)

test_loss, test_acc = model.evaluate(test_dataset, test_labels,
verbose=2)
print("\nTest accuracy:", test_acc, test_loss)
```

```
32/32 - 0s - 2ms/step - accuracy: 0.9340 - loss: 0.1814

Test accuracy: 0.9340000152587891 0.18137790262699127
```

Ajuste de modelos de clasificación alternativos

Elige al menos un método de clasificación no basado en redes neuronales (p.ej. regresión logística, árboles de decisión, reglas de clasificación, random forest, SVM, etc).

In [19]:
```
# Código de ajuste del modelo de clasificación 1 (DecisionTreeClassifier)

model_1 = DecisionTreeClassifier()
model_1.fit(X_train, Y_train)
predictions_1 = model_1.predict(X_validation)

# Utilizo validación cruzada estratificada de 10 veces
kfold_1 = StratifiedKFold(n_splits=10, random_state=1, shuffle=True)
cv_results_1= cross_val_score(model_1, X_train, Y_train, cv=kfold_1, scoring='accuracy')
print('CART: %f (%f)' % (cv_results_1.mean(), cv_results_1.std()))
```

```
CART: 0.923000 (0.012237)
```

In [20]:
```
# Código de ajuste del modelo de clasificación 2 (AdaBoostClassifier)
base_model = DecisionTreeClassifier()

model_2 = AdaBoostClassifier(estimator=base_model, n_estimators=100, random_state=42, algorithm='SAMME')
model_2.fit(X_train, Y_train)
predictions_2 = model_2.predict(X_validation)

# Utilizo validación cruzada estratificada de 10 veces
kfold_2 = StratifiedKFold(n_splits=10, random_state=1, shuffle=True)
cv_results_2= cross_val_score(model_2, X_train, Y_train, cv=kfold_2, scoring='accuracy')
print('AdaBoost: %f (%f)' % (cv_results_2.mean(), cv_results_2.std()))
```

```
AdaBoost: 0.923000 (0.011336)
```

Construya un o dos párrafos con los principales hallazgos. Incluye una explicación de los parámetros que consideres relevantes en cada ejecución.

Vemos como todos los modelos tienen una exactitud muy alta, lo cual nos indica que se ha entrenado correctamente. Vemos como la red neuronal ha tenido un pico más alto de precisión, pero su rango de resultados también ha sido más amplio, bajando así el porcentaje de aciertos. Por otra parte, los modelos sin red neuronal se han comportado de manera muy eficiente ante el problema de clasificación, consiguiendo un poco más de exactitud el modelo 'AdaBoostClassifier'.

Comparación del desempeño de modelos

Muestra los resultados obtenidos por los diferentes algoritmos escogidos de forma gráfica y comparada/superpuesta.

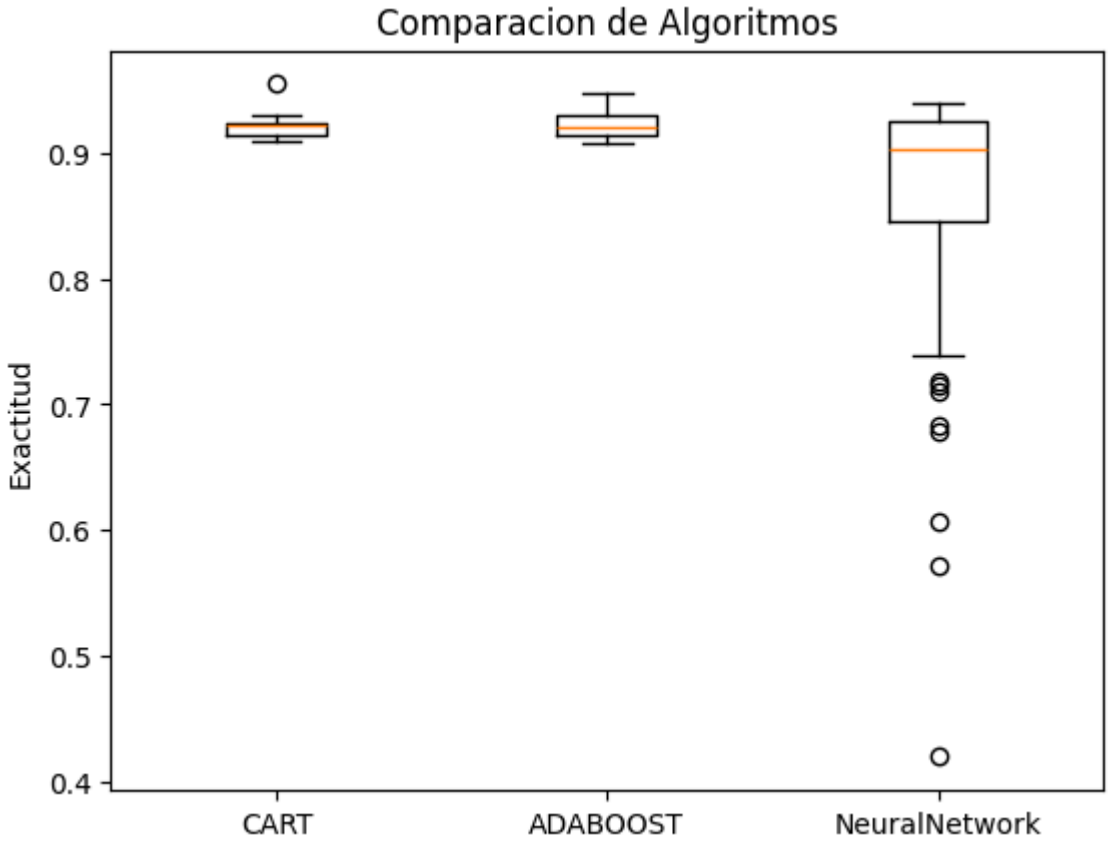
In [21]:
```
print('RedNeuronal: %.6f (%.6f)' % (test_acc,test_loss))
print('CART: %f (%f)' % (cv_results_1.mean(), cv_results_1.std()))
print('AdaBoost: %f (%f)' % (cv_results_2.mean(), cv_results_2.std()))
```

```
RedNeuronal: 0.934000 (0.181378)
CART: 0.923000 (0.012237)
AdaBoost: 0.923000 (0.011336)
```

In [22]:
```
# Código para mostrar la comparación de métricas de desempeño de las dos propuestas en tabla

# Código desempeño modelo de clasificación 1 (DecisionTreeClassifier)
print('Classification report DecisionTreeClassifier')
print(classification_report(Y_validation, predictions_1))
print('----------------')

# Código desempeño modelo de clasificación 2 (AdaBoostClassifier)
print('Classification report AdaBoostClassifier')
print(classification_report(Y_validation, predictions_2))
print('----------------')
```

```
Classification report DecisionTreeClassifier
              precision    recall  f1-score   support

        Good       1.00      0.99      0.99       404
   Hazardous       0.79      0.81      0.80       100
    Moderate       0.92      0.91      0.92       296
        Poor       0.79      0.80      0.79       200

    accuracy                           0.91      1000
   macro avg       0.87      0.88      0.88      1000
weighted avg       0.91      0.91      0.91      1000

----------------
Classification report AdaBoostClassifier
              precision    recall  f1-score   support

        Good       1.00      0.99      0.99       404
   Hazardous       0.80      0.83      0.81       100
    Moderate       0.92      0.92      0.92       296
        Poor       0.80      0.80      0.80       200

    accuracy                           0.92      1000
   macro avg       0.88      0.88      0.88      1000
weighted avg       0.92      0.92      0.92      1000

----------------
```

In [23]:
```python
# Código para mostrar la comparación de métricas de desempeño de las tres propuestas en gráfica

resultados = [cv_results_1, cv_results_2, cv_results_3]
nombres = ['CART','ADABOOST', 'NeuralNetwork']

# Comparacion de algoritmos
plt.boxplot(resultados, tick_labels=nombres)
plt.title('Comparacion de Algoritmos')
plt.ylabel('Exactitud')
plt.show()
```



Construya un párrafo con los principales hallazgos.

Como comentábamos al principio tienen mejor evaluación las clases 'Good' y 'Moderate' debido a su mayor cantidad de datos para el entrenamiento.

Por otra parte, aquí se puede apreciar como el algoritmo de redes neuronales empieza aprendiendo con fallos más grandes, y poco a poco logra estabilizarse en una zona donde ya obtiene una buena precisión.

Discusión de los resultados obtenidos y argumentos sobre cómo se podrían mejorar de dichos resultados

Realice en este espacio todo el análsis de resultados final incluyendo:

- Resultados comparados. Conclusiones objetivas y significantes con base a las diferentes métricas escogidas.
- Argumentos que describan con qué técnica se obtienen mejores resultados en base a las diferentes métricas que hayas escogido
- Explicación de cómo se podrían mejorar los resultados obtenidos por las redes neuronales, independientemente de que mejoren o no a los algoritmos no basados en redes neuronales.

Conclusiones

Una vez evaluados los tres modelos podemos decir que el mejor para este caso es el modelo 'ADABOOST', seguido de 'CART', y por último también con buenos resultados el modelo de red neuronal. El rendimiento de los tres es alto aún teniendo 9 variables de las que dependía la predicción. Aún así, creo que se podría mejorar la precisión de los modelos con una cantidad mayor de datos que evaluar, o bien si la variable objetivo dependiera de menos atributos, o incluso si estos tuvieran menos valores dentro de ellos, es decir, dividir por rangos las temperaturas, humedades... haciendo más simple el análisis.

Comparación

La métrica que ha resultado ser más exitosa es la de 'ADABOST', la cual se ha basado en un árbol de decisión. Este método se basa en la idea de combinar múltiples modelos débiles (como árboles de decisión) para formar un modelo fuerte con alta precisión.

Mejoras red neuronal

En este caso en particular, y observando y comparando los resultados, creo que la mejor opción sería, como he comentado antes, clasificar los atributos por rangos, ya que esto supondría una mayor precisión a la hora de predecir. Por otra parte, también sería buena opción ver si se podría quitar alguna variable de entrada de la que no dependa mucho, ya que el algoritmo cuantas menos variables que analizar tenga se comportará de mejor manera.